CSC1135 Secure Programming Lab 01

Darragh O'Brien B.Sc. Ph.D. FHEA
School of Computing
Dublin City University

Overview

In this introductory lab we will:

- Get the CSC1135 virtual machine up and running
- Invoke some system calls
- Examine the fork system call
- Solve a race condition in a multithreaded program
- Look at the use of function pointers in C
- Solve another race condition in a multithreaded program

A. CSC1135 virtual machine

Throughout the course we will be working with a 32-bit virtual machine running in VirtualBox. Download the virtual machine from this Google Drive directory. The zipped image is approximately 3 GB in size. This VM comes configured to use an older (simpler) version of the `gcc` compiler.

Once you have downloaded it, unzip the VM. Next launch VirtualBox (install it if you haven't done so already). Navigate to where you unpacked the VM and boot it in VirtualBox. User names (and corresponding passwords) for the virtual machine are student and root.

To simplify things while getting started, we have disabled address space layout randomisation (ASLR) on the CSC1135 virtual machine (we will talk more about this operating system security feature later in the course but for now it is enough to know that ASLR is designed to confuse buffer overflow attacks).

Virtual machine FAQ

Q. What command do I run to unpack the VM?
A. `unzip CSC1135.zip`

Q. How can I backup work carried out on the virtual machine?
A. You can `sftp` it to your account on `student.computing.dcu.ie`.

Q. How do I disable|enable address space layout randomisation?
A. As root run `/sbin/sysctl kernel.randomize_va_space=0|2`.

Q. How do I disable ASLR permanently?
A. Add `kernel.randomize_va_space=0` to `/etc/sysctl.conf` and run the `sysctl -p` command.

Q. How do I confirm address space layout randomisation has been disabled?
A. Running `/usr/sbin/sysctl kernel.randomize_va_space` should show `kernel.randomize_va_space=0`.

B. System calls

Take a copy of the C source code in getpid.c. Open the file in your favourite editor (e.g. `gedit`) and have a read of it. Try to understand what it does. From the code we see that `getpid` is called. What does `getpid` do? It returns the process ID (a number) of the calling process (i.e. the process calling `getpid`). For a more detailed explanation enter the following at the prompt in a terminal window:

```
$ man getpid
```

From the C source code we build an executable called `getpid` using `gcc` (the GNU C Compiler) as follows:

```
$ gcc -o getpid getpid.c
```

Running the program a couple of times you should see output something like the following:

```
$ ./getpid
pid: 3129
$ ./getpid
pid: 3130
$ ./getpid
pid: 3131
```

To which process does this number refer? Why does the number keep changing?

As you did for `getpid`, look up `getppid` in the on-line manual pages. The `getppid` system call returns the process ID of the calling process's parent process. Add a call to `getppid` to your program. Make sure to print out the value returned by the call to `getppid`. Recompile the modified program and run it again. You should see something like this:

```
$ ./getpid
pid: 5561
ppid: 3735
$ ./getpid
pid: 5562
ppid: 3735
```

The value of `ppid` does not change from one run to the next. Why not? Can you work out which process `ppid` refers to? (Hint: enter the `ps` command (it lists processes) in the terminal window and see if you can spot the parent process ID.)

What are the system call numbers for `getpid` and `getppid`? To answer this question open this list. It lists the Linux system calls. Search for `getpid` and `getppid`.

Linux provides the `strace` utility for tracing all of the system calls made by a program. You can use it to see system calls as they happen or to print out a summary once program execution has terminated. To get a summary of the system calls made by your program do this:

```
$ strace -c ./getpid
```

Amongst others you should see the calls to `getpid` and `getppid`. If `strace` is missing install it by running the following as `root`:

```
# dnf install strace
```

C. Fork

Take a copy of fork.c. Compile and run it as shown above. It uses the `fork` system call to create a copy of the calling process: where we had a single running process now we have two processes executing the same program. The copy is a child of the process which called `fork`. The value returned by `fork` in the parent is the process ID of the child while the value returned in the child is zero. Check out the man page for `fork` if in doubt. Taking advantage of this information use an `if...else` statement to have the parent print out *I am the parent* and the child print out *I am the child* before they exit.

D. A race condition

Take a copy of the C source code in race1.c. Build an executable as follows:

```
$ gcc -o race1 race1.c -lpthread
```

Take a look at the source code and work out what the program is doing. Run the executable. Does it produce the answer you expect? Does it produce the same answer each time you run it? What is wrong with the program? Add the following where necessary in order to fix the problem:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

Run the program again to verify it produces the correct result.

E. Function pointers

We can see there is some duplication of code across threads in race1.c. Take a look at race4.c. It does the same job but passes to each thread a pointer to the function to be applied by that thread. Fix this version in the same way you fixed the previous one.

F. Another race condition

Take a copy of the C source code in race2.c. Build an executable as follows:

```
$ gcc -o race2 race2.c -lpthread
```

Take a look at the source code and work out what the program is doing. Run the executable. Does it produce the answer you expect? Does it produce the same answer each time you run it? What is wrong with the program?

Look up the man page for `pthread_join` and use it to solve the problem with the program. Run the program again to verify it produces the correct result.